On the Reusable Specification of Non-functional Properties in DSLs

Francisco Durán[1], Steffen Zschaler[2], and Javier Troya[1]

[1] Departamento de Lenguajes y Ciencias de la Computación
Universidad de Málaga
{duran,javiertc}@lcc.uma.es
[2] Department of Informatics
King's College London
szschaler@acm.org

Abstract. Domain-specific languages (DSLs) are an important tool for effective system development. They provide concepts that are close to the problem domain and allow analysis as well as generation of full solution implementations. However, this comes at the cost of having to develop a new language for every new domain. To make their development efficient, we must be able to construct DSLs as much as possible from reusable building blocks. In this paper, we discuss how such building blocks can be constructed for the specification and analysis of a range of non-functional properties, such as, for example, throughput, response time, or reliability properties. We assume DSL semantics to be provided through a set of transformation rules, which enables a range of analyses based on model checking. We demonstrate new concepts for defining language modules for the specification of non-functional properties, show how these can be integrated with base DSL specifications, and provide a number of syntactic conditions that we prove maintain the semantics of the base DSL even in the presence of non-functional–property specifications.

1 Introduction

Domain-specific languages (DSLs) are an important tool for reaping the proposed benefits of model-driven engineering [1]. DSLs are languages based on concepts closer to the problem domain than the technical solution. They are, therefore, a good way to allow domain-experts, who may lack programming skills, to construct or participate in constructing substantial parts of new systems. In addition, because much more knowledge of the domain is available when interpreting statements in a DSL, it is possible to provide much more extensive code generation; this can enable complete generation of running systems from a relatively simple DSL-based model [2]. However, for DSLs to be effective, they may need to be implemented for very narrow domains [1], which implies that a large number of DSLs needs to be implemented. This requires highly efficient techniques for developing new DSLs, ideally based on an ability to reuse and compose partial languages for new domains.

In the design of software systems, many researchers distinguish between functional and non-functional properties (NFPs)—also sometimes referred to as extra-functional properties or quality of service. While functional properties are constraints on *what* the software system does, NFPs are constraints on *how* it does it—for example, how much resources are used or how long it takes to process an individual request. NFPs are important for the overall quality of a system, so they clearly need to be taken into account throughout development. We need to be able to predict and analyse NFPs from an early stage of development, so as to avoid costly re-design or re-implementation at a later stage. When developing systems based on DSLs, these DSLs, consequently, need to include an ability to express and analyse relevant NFPs. However, the analysis of NFPs is difficult and usually requires substantial specialist expertise. Integrating an ability to specify NFPs into DSLs can substantially increase the effort required to build a DSL. In this paper, we propose a technique for allowing NFP specification to be encapsulated into reusable DSL components. This way, the burden of specifying the NFPs of DSLs is drastically reduced, and specialist expertise is mainly required when the language component is constructed. Developing new DSLs capable of specifying particular NFPs in the context of a particular domain then becomes a matter of weaving in the NFP's language component.

The *e-Motions* language and system allows the definition of visual DSLs and their semantics through in-place model-transformation rules, providing support for their analysis through simulation or model checking in Maude [3]. In [4], Troya, Rivera, and Vallecillo build on the ideas of the *e-Motions* framework [5, 6] to keep track of specific NFPs by adding auxiliary objects to DSLs. However, their approach still requires the NFP specification and analysis component to be redefined from scratch for every new DSL. In this paper we build on their work, but aim to modularise the NFP part into its own language component. To do so, we take inspiration from the work in [7] where Zschaler introduced the notion of *context models* to provide an interface between TLA^+ specifications of non-functional and functional properties. We will use parametrisation over meta-models to achieve a similar effect for our language components. Specifically, we present a formal framework for such language components, syntactic conditions for their consistency and proofs of these conditions. We also present a basic prototype implementing these ideas in the context of *e-Motions*. However, a full integration is not in the scope of this current paper.

While our prototype and original motivation are for the case of *e-Motions*, both our approach and formal framework are more general. They can be applied for any DSL specification whose semantics are based on model transformations. Moreover, while our work is clearly motivated from the need of modularising NFP specifications, the formal framework covers arbitrary conservative extensions of such DSLs, guaranteeing them to be *spectative* in the sense of [8].

The remainder of this paper is structured as follows: In Section 2, we discuss a detailed motivating example to explain the vision of what we would like to achieve. Section 3 then presents a formalisation of these ideas together with consistency conditions and sketches of their proofs (see [9] for additional details on this). Section 4 briefly discusses our initial prototype. Finally, Section 5 discusses related work followed by conclusions and an outlook to future work in Section 6.

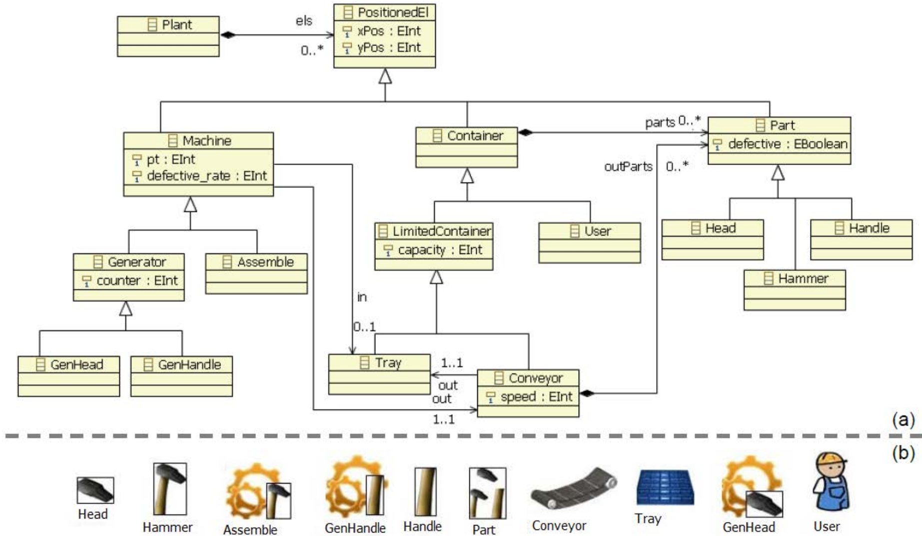


Fig. 1. Production line (a) metamodel and (b) concrete syntax (from [4])

2 Motivating Example

In this section, we present an example of what we want to achieve. This is based on work presented by Troya, Rivera, and Vallecillo in [4]. Their work defines DSLs from two parts: a meta-model of the language concepts and a set of transformation rules to specify the behavioural semantics of the DSL.

Figure 1(a) shows the metamodel of a DSL for specifying production-line systems, for producing hammers out of hammer heads and handles, which are generated in respective machines, and transported along the production line via conveyors and trays. As usual in MDE-based DSLs, this metamodel defines all the concepts of the language and their interconnections; in short, it provides the language's *abstract* syntax. In addition, a *concrete* syntax is provided. In the case of our example, this is sufficiently well defined by providing icons for each concept (see Figure 1(b)); connections between concepts are indicated through arrows connecting the corresponding icons.

Instances of this DSL are intended as token models [10]. That is, they describe a specific situation and not the set of all possible situations (as is the case, e.g., for class diagrams). The behavioural semantics of the DSL can, therefore, be given by specifying how models can evolve; that is, what changes can occur in a particular situation. This is specified through a set of model transformation rules. Figure 2 shows an example of such a rule. The rule consists of a left-hand side matching a situation before the execution of the rule and a right-hand side showing the result of applying the rule.[1] Specifically, this rule shows how a new hammer is assembled: a hammer generator `a`

[1] There are some other parts to the rule, but they are not relevant for our current discussion. For a more detailed discussion, please refer to material on *e-Motions* [5, 6].

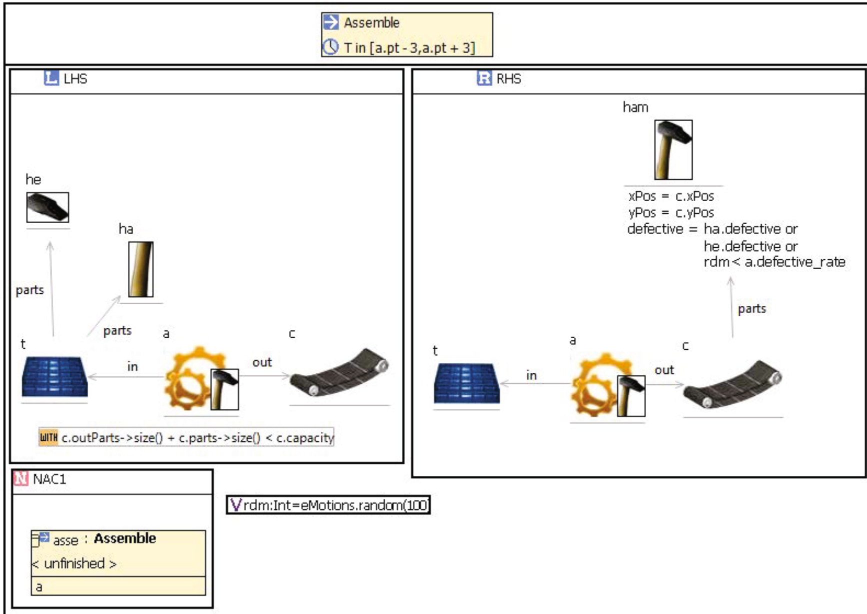


Fig. 2. `Assemble` rule indicating how a new hammer is assembled (from [4])

has an incoming tray of parts and is connected to an outgoing conveyor belt. Whenever there is a handle and a head available, and there is space in the conveyor for at least one part (specified by an OCL constraint in the left-hand side of the rule), the hammer generator can assemble them into a hammer. The new hammer is added to the `parts` set of the outgoing conveyor belt. The complete semantics of our production-line DSL is constructed from a number of such rules covering all kinds of atomic steps that can occur.[2]

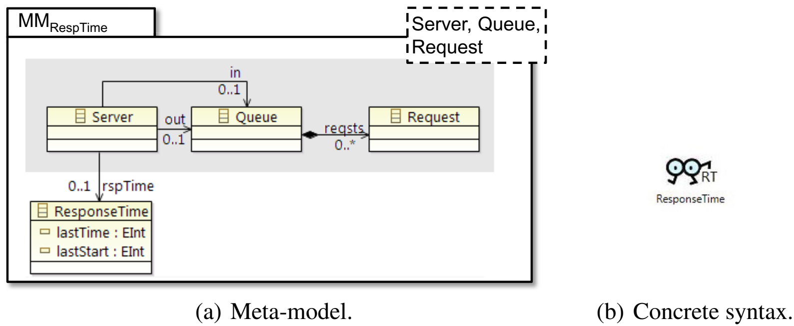
For production line systems, we are interested in a number of non-functional properties. For example, we would like to assess the throughput of the product line or how long it takes for a hammer to be produced.[3] We can achieve this by extending our DSL specification with observers [4]. Different from [4], here we suggest defining specification languages for observers entirely separately from any specific DSL. We will use the same mechanisms we used for defining the product line DSL to define a DSL that enables us to specify throughput or production time of systems.

Figure 3(a) shows the meta-model for a DSL for specifying production time. Two things should be noted about this meta-model:

---

[2] The complete specification of the Production Line example can be found at `http://atenea.lcc.uma.es/E-motions/PLSExample`.

[3] We use this property as an example here. Other properties can be defined easily in a similar vein as shown in [4] and on `http://atenea.lcc.uma.es/index.php/Main_Page/Resources/E-motions/PLSObExample`.



(a) Meta-model. (b) Concrete syntax.

Fig. 3. Meta-model and concrete syntax for response time observer

1. It defines no concept production time. Instead, it defines something called response time, which is a more generic concept. Production time is really only meaningful in the context of production systems. However, the general concept of response time covers this sufficiently well.
2. It is a parametric model (i.e., a model template). The concepts of `Server`, `Queue`, and `Request` and their interconnections are parameters of the meta-model, and they are shaded in grey for illustration purposes. We use them to describe in which situations response time can be specified, but these concepts will need to be mapped to concrete concepts in a specific DSL.

Figure 3(b) shows the concrete syntax for the response time observer object. Whenever that observer appears in a behavioural rule, it will be represented by that graphical symbol.

Figure 4 shows an example transformation rule defining the semantics of the response time observer. This states that if there is a server with an `in` queue and an `out` queue and there initially are some requests (at least one) in the `in` queue, and the `out` queue contains some requests after rule execution, the last response time should be recorded to have been equal to the time it took the rule to execute. Similar rules need to be written to capture other situations in which response time needs to be measured, for example, where a request stays at a server for some time, or where a server does not have an explicit `in` or `out` queue.

Note that the rule in Figure 4 looks different from the rule shown in Figure 2. This is because the rule is actually a rule transformation, while Figure 2 is a transformation rule. The upper part of Figure 4 (shaded in grey for illustration purposes) is a pattern or query describing transformation rules that need to be extended to include response-time accounting. The lower part describes the extensions that are required. So, in addition to reading Figure 4 as a 'normal' transformation rule (as we have done in the previous paragraph), we can also read it as a rule transformation stating: "Find all rules that match the shaded pattern and add `ResponseTime` objects to their left and right-hand sides as described." In effect, observer models become higher-order transformations [11].

As the rules in observer models are rule transformations, we can allow some additional concepts to be expressed. For example, Figure 4 uses multiplicities to express that

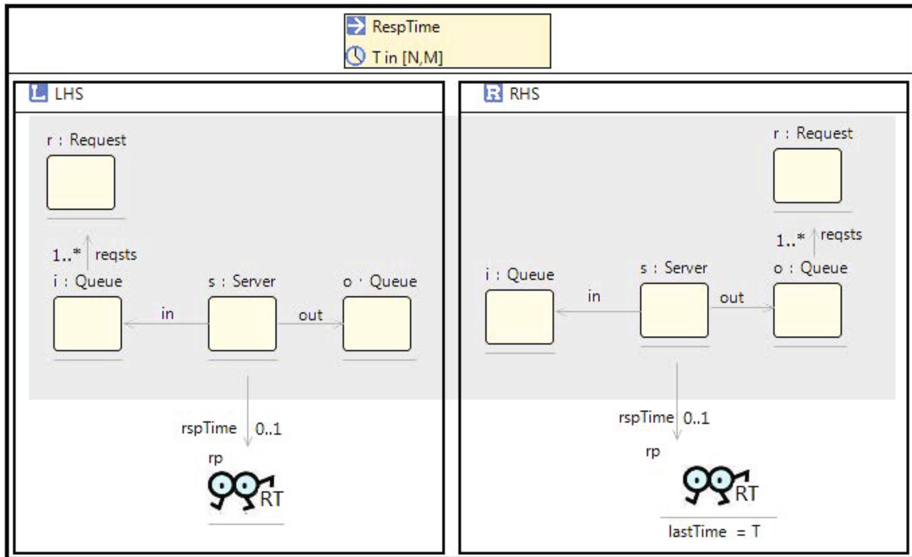


Fig. 4. Sample response time rule

there may be an arbitrary number of requests (but at least one) associated with a queue. This is not allowed in 'normal' transformation rules (there we need to explicitly show each instance). However, using multiplicities allows expressing patterns to be matched against transformation rules—a match is given by any rule that has the indicated number of instances in its left- or right-hand side.

To use our response-time language to allow specification of production time of hammers in our production-line DSL, we need to weave the two languages together. For this, we need to provide a binding from the parameters of the response-time meta-model (Figure 3(a)) to concepts in the production-line meta-model (Figure 1(a)). Specifically, we bind:

– **`Server` to `Assemble`** as we are interested in measuring response time of this particular machine;
– **`Queue` to `LimitedContainer`** as the `Assemble` machine is to be connected to an arbitrary `LimitedContainer` for queuing incoming and outgoing parts;
– **`Request` to `Part`** as `Assemble` only does something when there are `Parts` to be processed; and
– **Associations:**
  • The `in` and `out` associations from `Server` to `Queue` are bound to the corresponding `in` and `out` associations from `Machine` to `Tray` and `Conveyor`, respectively; and
  • The association from `Queue` to `Request` is bound to the association from `Container` to `Part`.

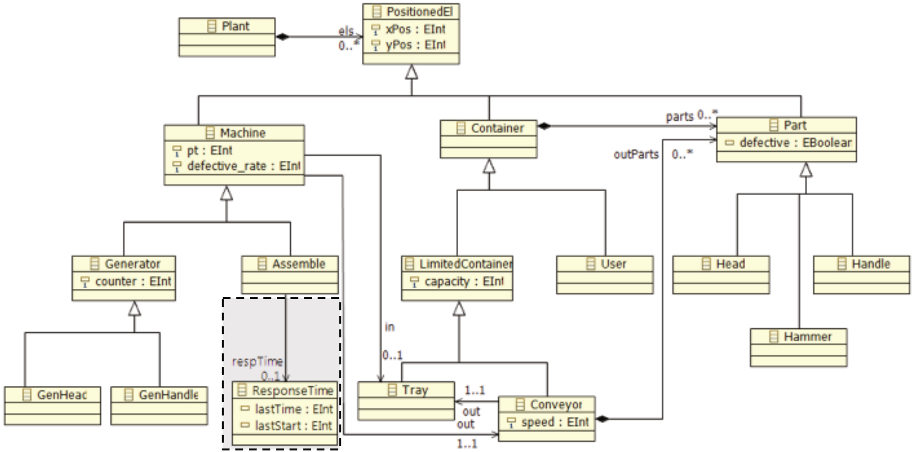


Fig. 5. Woven meta-model for measuring production time of the hammer assembler (highlighting added for illustration purposes)

Weaving the meta-models according to this binding produces the meta-model in Figure 5. The weaving process has added the `ResponseTime` concept to the meta-model. Notice that the weaving process also ensures that only sensible woven meta-models can be produced: for a given binding of parameters, there needs to be a match between the constraints expressed in the observer meta-model and the DSL meta-model. We will discuss this issue in more formal detail in Section 3.

The binding also enables us to execute the rule transformations specified in the observer language. For example, the rule in Figure 2 matches the pattern in Figure 4, given this binding: In the left-hand side, there is a `Server` (`Assemble`) with an in-`Queue` (`Tray`) that holds two `Requests` (`Handle` and `Head`) and an out-`Queue` (`Conveyor`). In the right-hand side, there is a `Server` (`Assemble`) with an in-`Queue` (`Tray`) and an out-`Queue` (`Conveyor`) that holds one `Request` (`Hammer`). Consequently, we can apply the rule transformation from Figure 4, which produces the rule shown in Figure 6. This rule is equivalent to what would have been written manually.

Clearly, such a separation of concerns between a specification of the base DSL and specifications of languages for non-functional properties is desirable. In the next section, we discuss the formal framework required for this and how we can distinguish safe bindings from unsafe ones.

3 Formal Framework

Graph transformation [12] is a formal, graphical and natural way of expressing graph manipulation based on rules. In graph-based modelling (and meta-modelling), graphs are used to define the static structures, such as class and object ones, which represent

Fig. 6. Result of weaving Figure 2 and Figure 4

visual alphabets and sentences over them. We formalise our approach using the typed graph transformation approach, specifically the Double Pushout (DPO) algebraic approach, with positive and negative application conditions [13]. Our graphs are, in particular, typed attributed graphs [14]. We however carry on our formalisation for weak adhesive high-level replacement (HLR) categories (see [15]).

The concepts of adhesive and (weak) adhesive HLR categories abstract the foundations of a general class of models, and comes together with a collection of general semantic techniques. Thus, e.g., given proofs for adhesive HLR categories of general results such as the Local Church-Rosser, or the Parallelism and Concurrency Theorem, they are automatically valid for any category which is proved an adhesive HLR category. This framework has been a break-through for the DPO approach of algebraic graph transformation, for which most main results can be proven in these categorical frameworks, and instantiated to any HLR system. One of these cases is the one of interest to us: the category of typed attributed graphs was proven to be an adhesive HLR category in [14].

In this section, we present a formal framework of what it means to define specification languages for non-functional properties separately to 'normal' DSLs, and in a way that can be reused across such DSLs. To this end, we will first abstract away from the concrete representation of languages and models in *e-Motions* [5, 6] that we have used in Section 2. Instead, we will formally represent the key elements of which such languages and models consist and the functions which are used to manipulate them.

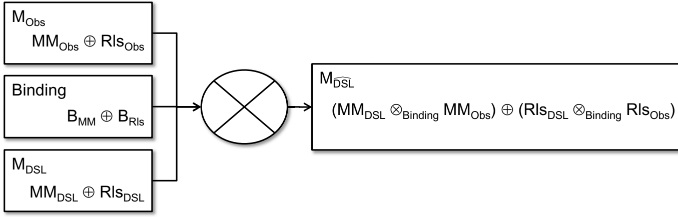


Fig. 7. Architecture of the formal framework

Figure 7 provides a graphical overview of the formal framework we are proposing. It can be seen that this consists of five parts:

1. M_{DSL}: The specification of a DSL (without any notion of non-functional properties);
2. M_{Obs}: The specification of a language for modelling non-functional properties of interest;
3. *Binding*: An artefact expressing how the parameters of M_{Obs} should be instantiated with concepts from M_{DSL} in order to weave the two languages;
4. $\otimes$: A function that performs the actual weaving; and
5. $M_{\widehat{DSL}}$: A DSL that combines the specification of some functionality (as per M_{DSL}) and some non-functional properties (as per M_{Obs}).

3.1 The Models Involved and Their Relationships

Following the algebraic graph transformation approach, a DSL can be seen as a typed graph grammar. A *typed graph transformation system* $GTS = (TG, P)$ consists of a type graph TG and a set of typed graph productions P. A *typed graph grammar* $GG = (GTS, S)$ consists of a typed graph transformation system GTS and a typed start graph S. A language is then defined by the set of graphs reachable from S using the transformation rules P.

Definition 1 (DSL). The specification M_X of a DSL X is given by a metamodel MM_X, representing the structural concepts of the language, and a set of transformation rules Rls_X, defining its behavioural semantics. ■

A metamodel is just a type graph, and a transformation rule associated to it is a graph production typed over the type graph provided by such metamodel.

The languages M_{DSL} and $M_{\widehat{DSL}}$ are DSL specifications. M_{Obs} is, essentially, also a normal DSL specification. Notice that we assume a single observer model M_{Obs} for each non-functional property. If we needed several of these properties, we could consider M_{Obs} to be the combination of the specifications of these non-functional properties, or we could iterate the process by instantiating $M_{\widehat{DSL}}$ once obtained with a second observers model $M_{Obs'}$ producing a resulting specification $M_{\widehat{\widehat{DSL}}}$, which could again be instantiated by another observers model $M_{Obs''}$, etc.

Although M_{Obs} is, essentially, a normal DSL specification, it is however parametrised and has a dual interpretation:

1. As a specification language for non-functional properties; and
2. As a higher-order transformation specification [11] expressing how DSLs need to be modified to enable the specification of a particular non-functional property.

Specifically, in the observer model M_{Obs}, we distinguish a parameter sub-model M_{Par} which specifies just enough information about real systems to define the semantics of the non-functional property, but not more. This parameter sub-model M_{Par} is a sub-model of M_{Obs} in the sense that MM_{Par} is a subgraph of MM_{Obs}, and each transformation rule in Rls_{Par} is a sub-rule of a rule in Rls_{Obs}.

This notion of sub-model and that of binding are captured by the general notion of DSL morphism, which can be defined as follows.

Definition 2 (DSL Morphism). Given DSL specifications M_A and M_B, a *DSL morphism* $M_A \to M_B$ is a pair (δ, ω) where δ is a meta-model morphism $MM_A \to MM_B$, that is, a mapping (or function) in which each class, attribute and association in the meta-model MM_A is mapped, respectively, to a class, attribute and association in MM_B such that

1. class maps in δ must be compatible with the inheritance relation, that is, if class C inherits from class D in MM_A, then $\delta(C)$ must inherit from class $\delta(D)$ in MM_B;
2. class maps and association maps in δ must be consistent, that is, the images of the extremes of an association K in MM_A must lead to classes associated by the association $\delta(K)$;
3. attribute maps and class maps in δ must be consistent, that is, given an attribute a of a class C, its image $\delta(a)$ must be an attribute of the class $\delta(C)$;

and where ω is a set of transformation rules such that for each rule $r_1 \in Rls_A$ there is a transformation rule $\sigma : r_1 \to r_2$ for some $r_2 \in Rls_B$.

Rules r_1 and r_2 in a rule map $\sigma : r_1 \to r_2$ must have the same time constraints (ongoing or atomic, same duration, softness, periodicity, etc.). ■

Definition 2 could be relaxed in several ways, e.g., condition 2 could be relaxed to allow superclasses of the images of the extremes of an association K to be related by its image $\delta(K)$; similarly for condition 3, since it could be that the attribute $\delta(a)$ is an attribute inherited from some superclass of $\delta(C)$. However, we leave these relaxations, and a study of their effects on the formal framework presented, to future work.

Note also that the role of the parameter model M_{Par} is useful for establishing the way in which the DSL's metamodel and rules are to be modified. The binding DSL morphism is key for it, since it says how the transformations are to be applied. An inclusion morphism $(\iota, \omega) : M_{Par} \hookrightarrow M_{Obs}$ can be seen as a transformation rule, with the binding indicating how such rule must be applied to modify the DSL system being instrumentalised. Specifically, the metamodel morphism $\iota : MM_{Par} \hookrightarrow MM_{Obs}$ indicates how the metamodel MM_{DSL} must be extended, and the family of higher-order transformations (HOT) rules $\sigma_i : r_{1,i} \hookrightarrow r_{2,i}$ in ω indicate how the rules in Rls_{DSL} must be modified. Notice that these rule transformations match those presented in Section 2:

A transformation rule like the one in Figure 4 is interpreted as a rule transformation where the parameter part (the shadowed sub-rule) is its left-hand side, and the entire rule its right-hand side.

Since M_{Par} is not intended for DSL specification, but only as constraints in the different transformations involved, i.e., for matching, we can enrich its expressiveness, for example, by allowing associations with multiplicity $1..n$ as in Figure 4. The left-hand side of this HOT rule can in this way match rules whose queues have 1, 2, or any number of request objects associated. Notice that this rule is woven with the `Assemble` rule in Figure 2, which has a head and a handle associated to its `in` tray. It could be seen as the inclusion and the binding happening on specific submodels, those defined by the concrete match. We do not consider this additional flexibility in the formalisation below. Notice however that the matches induce corresponding rules for which the formalisation does work.

3.2 Model Weaving

M_{Obs} and M_{DSL} are woven to produce a combined DSL, $M_{\widehat{DSL}}$. This weaving is encoded as a function $\otimes : M_{Obs} \times M_{DSL} \times \mathit{Binding} \to M_{\widehat{DSL}}$, which is graphically depicted in Figure 7. As indicated above, *Binding* is a DSL morphism, which expresses how the parameters of M_{Obs} should be instantiated with concepts from M_{DSL} in order to weave the two languages. Intuitively, $\otimes$ works in two stages:

1. *Binding stage.* In this stage, *Binding* is used to produce an instantiated version of M_{Obs} (and its parameter sub-model M_{Par}), $M_{Obs'} = (MM_{Obs'}, Rls_{Obs'})$, which is the result of replacing each parameter element $p \in MM_{Par}$ by a corresponding element from MM_{DSL} in M_{Obs} in accordance with *Binding*. The resulting $MM_{Obs'}$ is used to construct the output meta-model $MM_{\widehat{DSL}} = MM_{DSL} \uplus_{\mathit{Binding}} MM_{Obs'}$. The operator $\uplus_{\mathit{Binding}}$ stands for disjoint union, where elements related by *Binding* are identified and the rest are distinguished. Each rule $\sigma'_i : r'_{1,i} \hookrightarrow r'_{2,i}$ in $Rls_{Obs'}$ is the result of a similar replacement of a rule $\sigma_i : r_{1,i} \hookrightarrow r_{2,i}$ in Rls_{Obs}.
2. *Transformation stage.* In this stage, $Rls_{Obs'}$ is used to transform Rls_{DSL}. For each inclusion morphism $\sigma'_i : r'_{1,i} \hookrightarrow r'_{2,i}$, the corresponding rule $r \in Rls_{DSL}$ is identified and transformed according to σ'_i. This step produces $Rls_{\widehat{DSL}}$.

Note that the rules and appropriate matches to apply the HOT rules should be guided by *Binding*. Although there might be cases in which we can systematically apply the HOT rules on the rules in Rls_{DSL}, in general this is not the case. Note that each HOT rule defined by a rule in Rls_{Obs} may be applicable to different rules in Rls_{DSL}, and for each of them there might be more than one match. Although there might be many cases in which a partial binding might be enough, we however assume that the binding is complete.

The semantics of the weaving operation, informally described above, is provided by the pushout of DSL morphisms $M_{Par} \hookrightarrow M_{Obs}$ and $M_{Par} \hookrightarrow M_{DSL}$ in the category **DSL** of DSL specifications and DSL morphisms. Although the details of the pushout construction can be found in [9], we sketch it here. Given DSL morphisms $(\delta_1, \omega_1) : A \to B$ and $(\delta_2, \omega_2) : A \to C$, with DSLs $X = (MM_X, Rls_X)$ for $X = A$,

B, C, the pushout object of (δ_1, ω_1) and (δ_2, ω_2) is, up to isomorphism, the DSL specification $D = (MM_D, Rls_D)$, together with DSL morphisms $(\delta'_1, \omega'_1) : C \to D$ and $(\delta'_2, \omega'_2) : B \to D$, where $\delta'_1 : MM_C \to MM_D$ and $\delta'_2 : MM_B \to MM_D$ are the pushout of $\delta_1 : MM_A \to MM_B$ and $\delta_2 : MM_A \to MM_C$, and Rls_D is the disjoint union of those rules in Rls_B and Rls_C that are not targets of rules in Rls_A, and the set of rules $\tilde{r}_A$ resulting from the amalgamation of rule morphisms $r_1^A \to r_2^B$ in ω_1 and $r_1^A \to r_2^C$ in ω_2 for all rules r_A in Rls_A. The rule injections from rules in Rls_B and Rls_C that are not targets of rules in Rls_A and the amalgamation-induced rule morphisms $r_2^B \to \tilde{r}_A$ and $r_2^C \to \tilde{r}_A$ characterise the family of rule transformations ω'_2 (resp., ω'_1).

$$\begin{array}{ccc} A & \xrightarrow{(\delta_2,\omega_2)} & C \\ {\scriptstyle (\delta_1,\omega_1)}\downarrow & po & \vdots\downarrow{\scriptstyle (\delta'_1,\omega'_1)} \\ B & \xrightarrow[(\delta'_2,\omega'_2)]{\cdots\cdots} & D \end{array}$$

3.3 Semantic Consistency

The construction of *Binding* as a binding morphism $(\delta, \omega) : M_{Par} \to M_{DSL}$ ensures basic syntactic consistency between the observer model and the DSL model to be woven. However, it does not ensure semantic consistency. We could, for instance, specify a deadlock behaviour in the observers, changing the behaviour of the system DSL after the weaving. While it will likely not be possible to provide sufficient conditions for binding validity (see [7, pp. 9–11] for a discussion of the reasons), we should be able to provide at least some necessary conditions. As a minimum, we require that the extension be *conservative*, not changing the very nature and behaviour of the original DSL, namely:

$$M_{\widehat{DSL}}|_{MM_{DSL}} \cong M_{DSL} \tag{1}$$

We use $M_{\widehat{DSL}}|_{MM_{DSL}}$ to denote the language specification that results from removing any non-MM_{DSL} elements from the meta-model and rule set of $M_{\widehat{DSL}}$. Essentially, we mean to say that adding observers does not change the basic structure and behaviour defined by M_{DSL}. This is the typical condition one would expect in this kind of situations, and has been established in many different contexts before—perhaps first in [16].

Condition (1) is too hard to check directly, and therefore we need simpler, if possible syntactic, conditions implying it. If we break (1) down into conditions for the meta-model and the rule component of $M_{\widehat{DSL}}$, we get the following two conditions:

$$MM_{\widehat{DSL}}|_{MM_{DSL}} \cong MM_{DSL} \tag{2}$$

$$\Pi\left(Rls_{\widehat{DSL}}\right)|_{MM_{DSL}} \cong_{\mathit{stuttering}} \Pi\left(Rls_{DSL}\right) \tag{3}$$

We have again used the restriction operator $|_{MM_{DSL}}$, although in this case applied both to meta-models and traces, but with the same effect, namely, removing any non-MM_{DSL} elements (both from the meta-model $MM_{\widehat{DSL}}$ and rules in $Rls_{\widehat{DSL}}$). $\Pi\left(Rls_X\right)$ denotes the set of behaviours (possible executions, or traces) modelled by the transformation rules of a DSL X; that is, the set of all (potentially infinite) traces of model states

as rewritten by these transformation rules. For traces, we use $\cong_{stuttering}$ to explicitly state that the traces in $\Pi\left(Rls_{\widehat{DSL}}\right)|_{MM_{DSL}}$ may in fact have more steps than those in $\Pi\left(Rls_{DSL}\right)$, but because of the restriction down to MM_{DSL}, these remaining extra steps should all be identities (stuttering steps).

The above conditions (2) and (3) could only be checked by performing the weaving and analysing the result. However, both the weaving and the checking on the resulting specification are potentially expensive. Instead, we want to check the safety of an observer model and a binding morphism simply by looking at the models themselves without having to perform the weave. We are, therefore, looking for conditions on M_{Obs} and the binding morphism, if possible syntactic, so that they can be automated, or at least, performed once and for all. We in fact claim that analysing M_{Obs}, and simple syntactic conditions on the parameter inclusions $M_{Par} \hookrightarrow M_{Obs}$ and on the instantiating binding $(\delta, \omega) : M_{Par} \to M_{DSL}$ are sufficient to imply the satisfaction of (2) and (3).

We first discuss conditions for the structural part encoded in MM_{Obs}, and then discuss the behavioural semantics.

Structural Conditions. In any adhesive category, the pushout of a monomorphism along any map is a monomorphism [17, Proposition 2.1]. Therefore, since $MM_{Par} \to MM_{Obs}$ is a monomorphism, the induced morphism $MM_{DSL} \to MM_{\widehat{DSL}}$ is also a monomorphism. Notice that in addition to new classifiers, attributes and associations involving observers, new attributes for classes in MM_{Par} and new associations between classes in MM_{Par} may be introduced in MM_{Obs}. This might be convenient for modelling some NFPs and does not cause problems from a semantic point of view.

Behavioural Conditions. We need to ensure that adding observers to a DSL specification does not prevent any behaviour of any instance of that DSL that was previously allowed, and, moreover, that no new behaviours are added.

It can be shown that we can have (3) by imposing a similar condition on the morphism $M_{Par} \to M_{Obs}$. More precisely, it can be shown that

$$\Pi\left(Rls_{Obs}\right)|_{MM_{Par}} \equiv_{stuttering} \Pi\left(Rls_{Par}\right) \tag{4}$$

implies

$$\Pi\left(Rls_{\widehat{DSL}}\right)|_{MM_{DSL}} \equiv_{stuttering} \Pi\left(Rls_{DSL}\right)$$

We still need methods for checking (4). The formalisation of the construction in [9] suggests some ideas in this direction, but we leave it as future work.

4 A Prototypical Implementation

For the implementation of our prototype we have used ATL [18], a hybrid model transformation domain specific language that contains a mixture of declarative and imperative constructs. ATL transformations are unidirectional, operating on read-only source

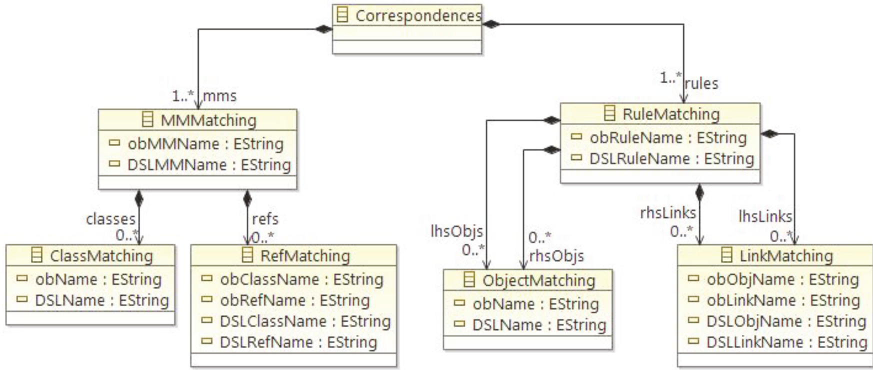


Fig. 8. Correspondences meta-model

models and producing write-only target models. During the execution of a transformation, source models may be navigated but changes are not allowed. Target models cannot be navigated.

Following the proposal presented in Figure 7, we have split the binding process in two ATL transformations: one for weaving the meta-models, MM_{DSL} and MM_{Obs}, and another one for weaving the behavioural rules, Rls_{DSL} and Rls_{Obs}. In our example, the former produces the meta-model shown in Figure 5, while the latter produces the woven rule depicted in Figure 6 (plus the remaining rules in Rls_{DSL}). For the remainder of this section, let us clarify that by *binding* we mean the relations established between two models. As for *correspondence(s)* and *matching(s)*, we use them indistinctly when we refer to one or more specific relations among the concepts in both models (either DSL and observer meta-models or DSL and observer behavioural rules).

The binding between M_{DSL} and M_{Obs} is given by a model that conforms to the correspondences meta-model shown in Figure 8. Thus, both bindings, between meta-models and between behavioural rules, are given in the same model. For the binding between meta-models (Figures 1(a) and 3(a) in our example), we have the classes `MMMatching`, `ClassMatching` and `RefMatching` that specify it. We will have one object of type `MMMatching` for each pair of meta-models that we want to weave. In our example, we have one object of this type, and its attributes contain the names of the meta-models to weave. Objects of type `MMMatching` contain as many `classes` (objects of type `ClassMatching`) as there are correspondences between classes in both meta-models. Each object of type `ClassMatching` stores the names of the classes in both meta-models that correspond. We have three objects of this type, as described in Section 2. Regarding the objects of type `RefMatching`, contained in the `refs` reference from `MMMatching`, they store the matchings between references in both meta-models. Attributes `obClassName` and `DSLClassName` keep the names of the source classes, while `obRefName` and `DSLRefName` contain the names of the references. Once again, and as described in Section 2, there are three objects of this type in our example.

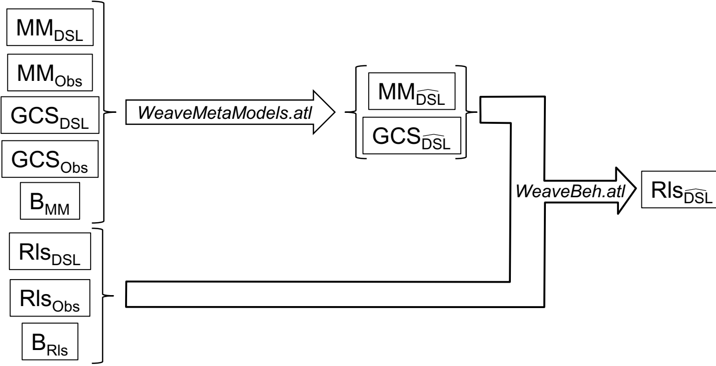


Fig. 9. Transformations schema

Regarding the binding between rules (Rls_{DSL} and Rls_{Obs}), there is an object of type `RuleMatching` for each pair of rules to weave, so in our example there is only one. It contains the names of both rules (`Assemble` and `RespTime`). Objects of types `ObjectMatching` and `LinkMatching` contain the correspondences between objects and links, respectively, in the rules. Concretely, our correspondence models differentiate between the bindings established between left- and right-hand side in rules, as we describe later. In our behavioural rules described within *e-Motions*, which conform to the `Behavior` meta-model (presented in [5]), the objects representing instances of classes are of type `Object` and they are identified by their `id` attribute, and the links between them are of type `Link`, identified by their name, input and output objects. Similar to the binding between meta-models, objects of type `ObjectMatching` contain the identifier of the objects matching, and instances of `LinkMatching` store information about matchings between links (they store the identifier of the source classes of the links as well as the name of the links). The correspondences between rules `Assemble` and `RespTime` are those described at the end of Section 2.

A detailed documentation of the weaving process, performed by two ATL transformations, is available in [19]. Here, we limit ourselves to a high-level overview of the transformation architecture. As shown in Figure 9, we have split the overall weaving function into two model transformations, one for weaving the metamodels and the other for weaving the rules. Apart from the models already presented in Figure 7, `GCS` models (graphical concrete syntax) also take part in the transformations. They store information about the concrete syntax of DSL and observer models. Both transformations work in two stages to ensure the original DSL semantics are preserved. First, they copy the original DSL model into the output model. Second, any additions from the observer model are performed according to the binding information from the weaving model.

The first transformation, named `WeaveMetaModels.atl`, deals with the weave of meta-models and `GCS` models. In the first step, the transformation copies both models from the DSL into the output models. Next, it decorates the models created with the concepts from the observer meta-model and `GCS` model. Regarding the output meta-model, it adds the classes, references and attributes representing observers. In our example this

means inserting the ResponseTime class, adding its attributes and establishing the respTime reference among Assemble and ResponseTime classes. As for the output GCS file, it means adding all the necessary data regarding the concrete syntax of the ResponseTime class.

Between its inputs, the second transformation, WeaveBeh.atl, takes the models produced by the first transformation. It performs in a similar way. The first step is to copy all those rules from Rls_{DSL} in the output model with the behavioural rules. Next, those rules having correspondences with rules in Rls_{Obs} are decorated with observer objects, links and attributes.

5 Related Work

We discuss related work in two areas: modelling of non-functional properties and modular language definition.

5.1 Modelling of Non-Functional Properties

Modelling and analysis of non-functional properties has been an active research area for a substantial amount of time already. Our work is related to other work aiming to support specification of a wide range of non-functional properties—for example, languages such as QML [20], CQML [21], CQML$^+$ [22], or SLAng [23]. These languages take a meta-modelling approach to the specification of non-functional properties in a two-step process: In a first step, modellers specify *non-functional characteristics*—for example, performance. These characteristics are then used in a second step to express constraints over application models; that is, non-functional properties. This is similar to our approach: An observer model M_{Obs} effectively defines a non-functional characteristic. A woven DSL $M_{\widehat{DSL}}$ can then be used to model non-functional properties. The approaches mentioned above differ in their amount of formal rigor (increasing from QML to CQML$^+$ and SLAng) and the type of systems they support (all except SLAng are aimed at component-based systems; SLAng is meant for service-based systems). They typically do not provide extensive support for analysis of the models created.

More formal renderings of these concepts can be found in [16] and [7]. The former presents a formal encoding of real-time properties using so-called history-determined variables, which are then used to model non-functional characteristics that depend on time. [7] extends this to a formal framework for specifying non-functional properties of component-based systems. While these approaches can potentially enable proofs of non-functional properties, it is not clear how well they are suited to predictive analysis of system properties—for example through simulation.

The approach by Troya and Vallecillo [4] aims to address this issue by providing a specification based on observers and transformations. This enables predictive analysis through simulation based on an encoding in *e-Motions* [5, 6], which is translated into Maude. However, their approach requires the details of a non-functional characteristic to be redefined completely for each DSL. Our proposal is an extension of this work

using ideas from [7, 16] to separate the specification of non-functional characteristics from that of the functional behavioural semantics of a DSL.

5.2 Modular Languages, Models, and Transformations

We propose to weave two language definitions: One language enables the (abstract) specification of a set of non-functional properties while the second language focuses entirely on specifying relevant behaviours in a particular domain. Below we briefly review some related work in the general area of modular definition of languages, models, and transformations. We discuss selected related work in three areas:

1. Modular definition of languages;
2. Modular definition of models; and
3. Modular definition of model transformations.

Modular Definition of Languages. There is a large body of work on modularly defining computer languages. Most of this work (e.g., [24–26]) deals with textual languages and in particular with issues of composing context-free grammars. While the general idea of language composition is relevant for our work, this specific strand of research is perhaps less related and will, therefore, not be discussed in more detail.

For languages based on meta-modelling, there is much less research on language composition. Much of the work on model composition (see next sub-section) is of course of relevance as meta-models are models themselves. Christian Wende's work on role-based language composition [27] is an approach that specifically addresses the modularisation of meta-models. For a language module, Wende's work allows the definition of a composition interface by allowing language designers to use two types of meta-model concepts: meta-classes and meta-roles. Meta-classes are used as in normal meta-modelling to express the core meta-model concepts. Meta-roles are like meta-classes, however they actually represent concepts to be provided by another language—including definitions of operations and attributes, which are left abstract in the meta-role. Meta-roles are, thus, similar to our use of meta-model parameters in MM_{Obs}. However, Wende's work uses meta-class operations to provide an operational view on language semantics, while we use model transformations to encode language semantics.

Modular Modelling. Our notation for expressing parametrised meta-models is based on how UML expresses parametrised models. Similar notations have been used in aspect-oriented modelling (AOM) approaches—for example, Theme/UML [28] or RAM [29]. More generally, our language composition technique is based on the notion of model weaving from AOM. Theme/UML, RAM, or Reuseware [30] are examples of aspect-oriented modelling techniques, which are asymmetric [31]; that is, they make a distinction between a base model and an aspect model (the model that is parametrised) that is woven into the base model. This is also true of our approach: M_{DSL} is the base model and M_{Obs} is the model that is woven into it. There is an alternative approach to AOM that is more symmetric and considers all models to be woven as equal. This is typically based on identifying corresponding elements in different models and merging these. Examples are UML package merge or signature-based merging [32]. Most types

of AOM also consider syntactic weaving only, disregarding the semantics of the modular models. In contrast, we explicitly consider the model semantics and povide formal notions ensuring that the composition does not restrict the set of behaviours modelled in the base DSL.

Modular Model Transformations. The semantics of the languages we are discussing are expressed using model transformations. As such, work on modularising model transformations is of relevance to our work. Generally, this work can be distinguished into work on external and on internal modularisation of model transformations: The former considers a complete model transformation as the unit of modularity, while the latter aims to provide modularity inside individual transformations [33]. As we are modifying the internals of the base transformation by adding in detail described in the observer transformation rules, our approach is an *internal* modularisation technique. Nonetheless, ideas from external composition approaches are of interest to us. In particular, the work on model typing and reusable model transformations presented in [34] shows how the set of meta-model concepts effectively used by a model transformation can be computed and how this can be used to make the transformations more reusable. This is similar to the way in which we use the parametrised part of MM_{Obs} to make the observer transformation rules more reusable and to adapt them to different DSLs.

6 Conclusions and Outlook

We have presented a formal framework for language components for the specification of non-functional properties (NFPs) in domain-specific languages (DSLs). Specifically, this enables language designers to encapsulate the semantics of particular NFPs in a reusable language specification that can be woven into a base DSL specification to produce a DSL that also enables the modelling and analysis of that particular NFP in the context of a specific domain. We have presented conditions for the consistency of such language components; in particular these ensure that weaving a language component with a DSL does not add neither remove valid behaviours from the semantics of any expressions in that DSL.

Our work makes a number of assumptions about the structure of the base DSL as well as about the NFPs to be specified. In the future, we aim to reduce these assumptions to provide a more general framework for the specification of NFPs in DSLs. Most importantly, we will further study the cases where there is no simple alignment between Rls_{Obs} and Rls_{DSL}. This will require more powerful pattern-expression constructs in $Rls_{Obs}|_{MM_{Par}}$ and a more complex weaving algorithm that allows observer rules to be bound to multiple DSL rules and vice versa. Our current formalisation also does not consider the effect of well-formedness rules defined for any of the DSLs involved, although their addition should be relatively straightforward.

Acknowledments. We would like to thank Antonio Vallecillo for fruitful discussions throughout the work on this paper, and to Fernando Orejas for his collaboration in the development of the formalisation of the proposal. This work has been partially supported by Spanish Government Project TIN2011-23795.

References

1. Hutchinson, J., Whittle, J., Rouncefield, M., Kristoffersen, S.: Empirical assessment of MDE in industry. In: Taylor, R.N., Gall, H., Medvidovic, N. (eds.) Proc. 33rd Int'l Conf. on Software Engineering (ICSE 2011), pp. 471–480. ACM (2011)
2. Hemel, Z., Kats, L.C.L., Groenewegen, D.M., Visser, E.: Code generation by model transformation: A case study in transformation modularity. Software and Systems Modelling 9(3), 375–402 (2010); Published on-line first at www.springerlink.com
3. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.: All About Maude - A High-Performance Logical Framework. LNCS, vol. 4350. Springer, Heidelberg (2007)
4. Troya, J., Rivera, J.E., Vallecillo, A.: Simulating domain specific visual models by observation. In: Proc. 2010 Spring Simulation Multiconference (SpringSim 2010), pp. 128:1–128:8. ACM, New York (2010)
5. Rivera, J.E., Durán, F., Vallecillo, A.: A graphical approach for modeling time-dependent behavior of DSLs. In: Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing, VL/HCC 2009, pp. 51–55. IEEE (2009)
6. Rivera, J.E., Durán, F., Vallecillo, A.: On the Behavioral Semantics of Real-Time Domain Specific Visual Languages. In: Ölveczky, P.C. (ed.) WRLA 2010. LNCS, vol. 6381, pp. 174–190. Springer, Heidelberg (2010)
7. Zschaler, S.: Formal specification of non-functional properties of component-based software systems: A semantic framework and some applications thereof. Software and Systems Modelling (SoSyM) 9, 161–201 (2009)
8. Katz, S.: Aspect Categories and Classes of Temporal Properties. In: Rashid, A., Akşit, M. (eds.) Transactions on AOSD I. LNCS, vol. 3880, pp. 106–134. Springer, Heidelberg (2006)
9. Durán, F., Orejas, F., Zschaler, S.: Behaviour protection in modular rule-based system specifications (submitted for publication, 2012)
10. Kühne, T.: Matters of (meta-) modeling. Software and Systems Modeling 5, 369–385 (2006)
11. Tisi, M., Jouault, F., Fraternali, P., Ceri, S., Bézivin, J.: On the Use of Higher-Order Model Transformations. In: Paige, R.F., Hartman, A., Rensink, A. (eds.) ECMDA-FA 2009. LNCS, vol. 5562, pp. 18–33. Springer, Heidelberg (2009)
12. Rozenberg, G. (ed.): Handbook of Graph Grammars and Computing by Graph Transformations. Foundations, vol. 1. World Scientific (1997)
13. Ehrig, H., Ehrig, K., Habel, A., Pennemann, K.H.: Theory of constraints and application conditions: From graphs to high-level structures. Fundamenta Informaticae 74(1), 135–166 (2006)
14. Ehrig, H., Prange, U., Taentzer, G.: Fundamental Theory for Typed Attributed Graph Transformation. In: Ehrig, H., Engels, G., Parisi-Presicce, F., Rozenberg, G. (eds.) ICGT 2004. LNCS, vol. 3256, pp. 161–177. Springer, Heidelberg (2004)
15. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamentals of Algebraic Graph Transformation. Springer (2005)
16. Abadi, M., Lamport, L.: An old-fashioned recipe for real time. ACM ToPLaS 16(5), 1543–1571 (1994)
17. Lack, S.: An embedding theorem for adhesive categories. Theory and Applications of Categories 25(7), 180–188 (2011)
18. Jouault, F., Allilaire, F., Bézivin, J., Kurtev, I.: ATL: a model transformation tool. Science of Computer Programming 72(1-2), 31–39 (2008)
19. Atenea: Reusable Specification of Observers (2012), `http://atenea.lcc.uma.es/index.php/Main_Page/Resources/ReusableObservers`

20. Frolund, S., Koistinen, J.: QML: A language for quality of service specification. Technical Report HPL-98-10, Hewlett-Packard Laboratories (1998)
21. Aagedal, J.Ø.: Quality of Service Support in Development of Distributed Systems. PhD thesis, University of Oslo (2001)
22. Röttger, S., Zschaler, S.: CQML$^+$: Enhancements to CQML. In: Bruel, J.M. (ed.) Proc. 1st Int'l Workshop on Quality of Service in Component-Based Software Engineering, pp. 43–56 (June 2003)
23. Skene, J., Lamanna, D.D., Emmerich, W.: Precise service level agreements. In: Proc. 26th Int'l Conf. on Software Engineering (ICSE 2004), pp. 179–188. IEEE Computer Society (2004)
24. Bravenboer, M., Visser, E.: Concrete syntax for objects: Domain-specific language embedding and assimilation without restrictions. In: Proc. 19th Annual ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2004), pp. 365–383. ACM Press (2004)
25. Bravenboer, M., Visser, E.: Parse Table Composition Separate Compilation and Binary Extensibility of Grammars. In: Gašević, D., Lämmel, R., Van Wyk, E. (eds.) SLE 2008. LNCS, vol. 5452, pp. 74–94. Springer, Heidelberg (2009)
26. Krahn, H., Rumpe, B., Völkel, S.: MontiCore: a framework for compositional development of domain specific languages. Int'l Journal on Software Tools for Technology Transfer (STTT) 12(5), 353–372 (2010)
27. Wende, C., Thieme, N., Zschaler, S.: A Role-Based Approach towards Modular Language Engineering. In: van den Brand, M., Gašević, D., Gray, J. (eds.) SLE 2009. LNCS, vol. 5969, pp. 254–273. Springer, Heidelberg (2010)
28. Carton, A., Driver, C., Jackson, A., Clarke, S.: Model-driven Theme/UML. Transactions on Aspect-Oriented Software Development (2008)
29. Kienzle, J., Abed, W.A., Klein, J.: Aspect-oriented multi-view modeling. In: Proc. 8th ACM Int'l Conf. on Aspect-Oriented Software Development (AOSD 2009), pp. 87–98. ACM (2009)
30. Heidenreich, F., Henriksson, J., Johannes, J., Zschaler, S.: On Language-Independent Model Modularisation. In: Katz, S., Ossher, H., France, R., Jézéquel, J.-M. (eds.) Transactions on AOSD VI. LNCS, vol. 5560, pp. 39–82. Springer, Heidelberg (2009)
31. Harrison, W.H., Ossher, H.L., Tarr, P.L.: Asymmetrically vs. symmetrically organized paradigms for software composition. Technical Report RC22685, IBM Research (2002)
32. Reddy, Y.R., Ghosh, S., France, R.B., Straw, G., Bieman, J.M., McEachen, N., Song, E., Georg, G.: Directives for Composing Aspect-Oriented Design Class Models. In: Rashid, A., Akşit, M. (eds.) Transactions on AOSD I. LNCS, vol. 3880, pp. 75–105. Springer, Heidelberg (2006)
33. Kleppe, A.G.: 1st European workshop on composition of model transformations (CMT 2006). Technical Report TR-CTIT-06-34, Centre for Telematics and Information Technology, University of Twente (June 2006)
34. Sen, S., Moha, N., Mahé, V., Barais, O., Baudry, B., Jézéquel, J.M.: Reusable model transformations. Software and Systems Modeling (SoSyM), 1–15 (2010)